ARI Research Note 88-01

LispSEI: The Programmer's Manual

Hans Tallis
University of California at Irvine

for

Contracting Officer's Representative
Judith Orasanu

BASIC RESEARCH LABORATORY
Michael Kaplan, Director

U. S. Army

Research Institute for the Behavioral and Social Sciences

January 1988

U. S. ARMY RESEARCH INSTITUTE

FOR THE BEHAVIORAL AND SOCIAL SCIENCES

A Field Operating Agency under the Jurisdiction of the

Deputy Chief of Staff for Personnel

EDGAR M. JOHNSON
Technical Director

WM. DARRYL HENDERSON
COL, IN
Commanding

Research accomplished under contract
for the Department of the Army

University of California at Irvine

Technical review by

Dan Ragland

Accession For	
NTIS GRA&I	X
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ARI Research Note 88-01	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) LispSEI: The Programmer's Manual		5. TYPE OF REPORT & PERIOD COVERED Interim Report January 86 - January 87 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Hans Tallis		8. CONTRACT OR GRANT NUMBER(s) MDA903-85-C-0324
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Information and Computer Science, University of California at Irvine Irvine, CA 92717		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 2Q161102B74F
11. CONTROLLING OFFICE NAME AND ADDRESS U.S. Army Research Institute for the Behavioral and Social Sciences, 5001 Eisenhower Avenue, Alexandria, VA 22333-5600		12. REPORT DATE January 1988 13. NUMBER OF PAGES 18
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) - -		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE - -

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

- -

18. SUPPLEMENTARY NOTES

Judith Orasanu, contracting officer's representative

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

World Modeler
Computer Models
Artificial Intelligence
Lisp
SEI

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This research note provides a listing of the lowest level Lisp functions, as well as a more precise definition of the capabilities of the different layers of the SEI. It also separates filtering from basic copying operations, shows how to hide the C code within the SEI, and cleans up some other unnecessary hair in the current SEI implementation.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

LispSEI: The Programmer's Manual

Hans Tallis

The Purpose of LispSEI and This Document is to:

1. Hide, once and for all, all C code within the SEI.
2. Provide a convenient listing of lowest level Lisp functions accessing and changing state in the SEI.
3. Provide a more precise definition of the capabilities of the different layers of the SEI. Currently the functions provided via FRANZSELL (for example) represent a mix of simple data copying operations (for example, passing the list of world objects on up to the organism) as well as more intelligent operations (the multi-step implementation of "turn" for example). LispSEI will provide only the simple data passing and control operations defined below, with the more complex operations being defined elsewhere in the SEI.
4. Separate out filtering operations from the basic copying of the world state. The filtering/massaging will be taking place in a number of library modules discussed elsewhere.
5. Clean up some other unnecessary hair present in the current SEI implementation.

Background

LispSEI is only meant to provide a minimal set of operations an organism could request. It is entirely possible that additional functionality could be added to the LispSEI later. N.B.: Currently it is not expected that Lisp-based organisms will be referencing LispSEI directly; rather, they will interact through the SEI library functions described in a separate document.

The organism interacts with the SEI in exactly three ways:

1. The organism receives state information from the SEI (objects in the world, emissions, and other miscellaneous information specific to the current SEI.)
2. The organism sends state-change requests to the SEI (emissions, motions of the organism, operations of the organism's parts, etc.) Note that this may be a superset of effector requests.
3. The organism sends control commands to the SEI (get world changes, send "ok to proceed," and so on).

Currently EFFECT.C implements functions of the second sort, while PERCEIVE.C implements functions of the first and third sort.

Changes

1. PERCEIVE.C will be split into two files, PERCEIVE.C and SEICONTROL.C, with the latter implementing control commands from the organism to the SEI.
2. The SEI will be written entirely in C and Lisp. LispSEI will be able to run under either Franz or Lucid Common Lisp (using the same source.)

3. The batch of functions defined in file LISPSEI (which implements LispSEI) will call functions in PERCEIVE.C, SEICONTROL.C, and EFFECT.C. The higher-level processing portions of the SEI (to be described in a future document) will call exclusively functions in LISPSEI.C.

4. RKLINK.C, SEILINK.SLISP, and SEIFACE.SLISP will all disappear. [We no longer write Spice Lisp organisms.]

5. Certain actions of PERCEIVE.C (*viz.* automatically copying the whole world of objects from C into Lisp) may no longer automatically be done (at the implementor's discretion).

6. The following will be the typical cycle for calls into LispSEI. The order is not too critical; in particular, steps *b* through *e* may occur in any order.

   a. **(Receive-Updated-World).**

   b. Request descriptions of objects by objNumber, one at a time. (The objNumber is a unique integer identifying that object within the world.) Alternately, individual fields of objects may be referenced.

   c. Request information from other senses, such as a list of things smelled, or a list of audible emissions, and so on.

   d. Request some effector actions, such as changing the object's motion vector, or emitting a sound, or swinging an arm around.

   e. Make other (illegal) changes to world objects. (Intuitively, illegal changes are those which a typical organism in a world could not easily carry out, such as changing the velocity of arbitrary objects at a distance.)

   f. **(Ok-To-Proceed)** to end a cycle.

The LispSEI thus provides essentially unlimited access to the world data structure and other state information.

Newly Defined Data Types and Their Accessing Functions

These are new to the LispSEI implementation. They are meant primarily to aid in the passing of data in a reasonably efficient way. The data types are just unguaranteed hints as to how these data packets might be implemented in LispSEI. *The user of LispSEI should only access these objects via the accessing functions described next to each data type.* Remember that the Set-ing functions operate only on the data object, and do not affect the actual objects, emissions, etc. in the world.

- `(setq example-Matrix-Type (make-array 4 4`
  `:element-type 'integer`
  `:initial-element 0))`

  (This is not strictly a data type, rather a template for one.)

  (Get-Matrix-Type-Elements-As-List *Matrix-Type*) returns the elements of a Matrix-Type object as a list in the order (Matrix-Type[0,0] Matrix-Type[0,1] Matrix-Type[0,2] ... MatrixType[3,3]).

(Get-Matrix-Type-Elements-As-Array *Matrix-Type*) returns a Lisp array.

- ```
  (declare-type Vector-Type
                (X 0.0 :type double)
                (Y 0.0 :type double)
                (Z 0.0 :type double))
  ```

  (Get-Vector-Type-X *Vector-Type*) returns a double.

  (Set-Vector-Type-X *value Vector-Type*) sets the *X* component of *Vector-Type* to be *value*.

  (Get-Vector-Type-Y *Vector-Type*) does the obvious thing.

  (Set-Vector-Type-Y *value Vector-Type*) does the obvious thing.

  (Get-Vector-Type-Z *Vector-Type*) does the obvious thing.

  (Set-Vector-Type-Z *value Vector-Type*) does the obvious thing.

- ```
  (declare-type Vertex-Type
                (X 0.0 :type double)
                (Y 0.0 :type double)
                (Z 0.0 :type double))
  ```

  (Get-Vertex-Type-X *Vertex-Type*) does the obvious thing.

  (Get-Vertex-Type-Y *Vertex-Type*) does the obvious thing.

  (Get-Vertex-Type-Z *Vertex-Type*) does the obvious thing.

  Set-ing functions are as for Vector-Type.

- ```
  (declare-type Apply-Type
                Force                                ;Vector-Type
                (Force-Time 0.0 :type double)
                Torque                               ;Vector-Type
                (Torque-Time 0.0 :type double)
                (Magnetic-Force 0.0 :type double)
                (Magnetic-Time 0.0 :type double))
  ```

  (Get-Apply-Type-Force *Apply-Type*) returns a Vector-Type, which can be further accessed via the accessors defined above.

  (Get-Apply-Type-Force-Time *Apply-Type*) does the obvious thing.

  (Get-Apply-Type-Torque *Apply-Type*) does the obvious thing.

  (Get-Apply-Type-Torque-Time *Apply-Type*) does the obvious thing.

  (Get-Apply-Type-Magnetic-Force *Apply-Type*) does the obvious thing.

  (Get-Apply-Type-Magnetic-Time *Apply-Type*) does the obvious thing.

  Set-ing functions are as for Vector-Type.

- ```
  (declare-type Color-Type
                (Red-Component 0 :type integer)
                (Green-Component 0 :type integer)
                (Blue-Component 0 :type integer))
  ```

  Accessors similar to those for Apply-Type. Set-ing functions are as for Vector-Type.

```
• (declare-type Taste-Type
            (Sweet 0 :type integer)
            (Sour 0 :type integer)
            (Bitter 0 :type integer)
            (Salty 0 :type integer))
```

Accessors similar to those for Apply-Type. Set-ing functions are as for Vector-Type.

```
• (declare-type Object-Type
            (Object-Number 0 :type integer)
            (Object-Name "" :type string)
            (Previous-Object 0 :type integer)
            (Next-Object 0 :type integer)
            (Part-Of 0 :type integer)
            (Part-Collision 0 :type integer)
            (Part-Magnetism 0 :type integer)
            Transform                          ;Matrix-Type
            (Scale-Factor 0.0 :type double)
            Location                           ;Vector-Type
            Rotation                           ;Vector-Type
            Velocity                           ;Vector-Type
            Rotational-Velocity                ;Vector-Type
            Acceleration                       ;Vector-Type
            Rotational-Acceleration            ;Vector-Type
            Forces                             ;Apply-Type
            (Buoyancy 0.0 :type double)
            Mass-Center                        ;Vector-Type
            (Mass 0.0 :type double)
            (Enclosure 0.0 :type double)
            (Kind-Of-Object 0 :type integer)
            (Composed-Of 0 :type integer)   ;for Complex
            (Changed-Below 0 :type integer)
            (Elasticity 0.0 :type double)   ;for Prim
            (Restitution 0.0 :type double)
            (Friction 0.0 :type double)
            Inside-Color                       ;Color-Type
            Outside-Color                      ;Color-Type
            (Outside-Surface-Texture 0 :type integer)
            (Inside-Surface-Texture 0 :type integer)
            Taste                              ;Taste-Type
            (Temperature 0.0 :type double)
            Axis                    ;Vector-Type; for Cylinder
            (Height 0.0 :type double)
            (Radius 0.0 :type double)
            Normal                  ;Vector-Type; for Polygon
            Vertices)               ; !! note:  a list of
                                    ;    Vertex-Types     !!
```

Accessors similar to those for Apply-Type, except:

(Get-Object-Type-Vertices *Object-Type*) returns a list of Vertex-Types.

Also, we have the function

(Get-Object-Slot *slot objNumber*) returns the value of the *slot* for the indicated *objNumber*. Slot names for this function are formed by prepending a colon to the name of the associated accessor function for Object-Type without the "*Get-*" prefix. Thus, (Get-Object-Slot

':Object-Type-Next-Object 7) returns the same thing as
(Get-Object-Type-Next-Object (Object 7)) returns.

Set-ing functions are as for Vector-Type.

- (declare-type Emission-Type

```
(Emission-ID 0 :type integer)
(Next 0 :type integer)
(Previous 0 :type integer)
(Emission-Type 0 :type integer)
Origin                              ;Vector-Type
(Time 0.0 :type double)
(Intensity 0 :type integer)
(Spread 0.0 :type double)
(Diffusion 0.0 :type double)
(Decay 0.0 :type double)
(Duration 0.0 :type double)
(Rep 0 :type integer)
(Bytes 0 :type integer)
Desc              ;a vector of size Bytes
```

Accessors similar to those for Apply-Type, except:

(Get-Emission-Type-Desc-As-List *Emission-Type*) returns the vector as a list.
(Get-Emission-Type-Desc-As-Array *Emission-Type*) returns the vector as a Lisp array.

Set-ing functions are as for Vector-Type.

- (declare-type Contact-Type

```
(Object1 0 :type integer);these 4 are ObjNumbers
(Object1-part  0 :type integer)
(Object2 0 :type integer)
(Object2-part  0 :type integer)
Point                               ; Vector-Type
Force                               ; Vector-Type
Magnetism-Flag)          ; boolean
```

Accessors similar to those for Apply-Type. Set-ing functions are as for Vector-Type.

A List of Functions

This is a list of the functions provided in Lisp by LISPSEI.

Receiving State Information

- **(Object-Head)** returns the objNumber of the head object for the world.
- **(Organism-Number)** returns the objNumber of the organism.
- **(Object *objNumber*)** returns an Object-Type corresponding to the world object identified as *objNumber*.
- **(Emission-Head)** returns the emitNumber of the head emission for the world.
- **(Emission *emitNumber*)** returns an Emission-Type corresponding to the world emission identified as *emitNumber*.

- **(Contacts)** returns a list of all Contact-Types for the world.
- **(Gravity)** returns a Vector-Type specifying the gravity vector.
- **(Time-Step)** returns the number of seconds in a cycle.
- **(World-Time)** returns the current world time.
- **(World-Steps)** returns the number of world steps in a cycle. Typically this is used only by the World Master.
- **(Directional-Light-Vector)** returns a Vector-Type specifying the directional light vector.
- **(Directional-Light-Color)** returns a Color-Type specifying the directional light color.
- **(Ambient-Light-Color)** returns a Color-Type specifying the ambient light color.
- **(Ambient-Temperature)** returns the ambient temperature.
- **(Viscosity)** returns the viscosity.
- **(World-Name)** returns a string naming the world.
- **(Trace-Name)** returns a string naming the trace of the world.
- **(Path-Name)** returns a string naming the directory into which SEI files will be written.
- **(Organism-Name)** returns a string naming the organism.
- **(Cycle-Number)** returns the number of cycles since the SEI started.

Sending State Change Requests

- **(Effect-Object-Velocity** *x y z objNumber***)** changes the velocity of the specified object.
- **(Effect-Object-Acceleration** *x y z objNumber***)** changes the acceleration of the specified object.
- **(Effect-Object-Rotational-Velocity** *x y z objNumber***)** changes the rotational velocity of the specified object.
- **(Effect-Object-Rotational-Acceleration** *x y z objNumber***)** changes the rotational acceleration of the specified object.
- **(Effect-Object-Torque** *x y z duration objNumber***)** applies a Torque for the specified duration to the specified object.
- **(Effect-Object-Force** *x y z duration objNumber***)** applies a force for the specified duration to the specified object.
- **(Effect-Object-Magnetism** *limb force duration***)** applies a specified magnetic force to the specified

limb for the specified duration.

- **(Effect-Object-Location** *x y z ObjNumber*) changes the location of the specified object.

- **(Effect-Object-Rotation** *x y z ObjNumber*) changes the rotation of the specified object.

- **(Delete-Object** *objNumber*) removes an object from the world.

- **(Create-Emission** *Emission-Type Origin Intensity Spread Diffusion Decay Duration Rep Bytes Desc*) requests the addition of an emission to the world. The types of the arguments are given by the description of the **Emission-Type** declare-type above. The *Origin* should be given as a list of three doubles. The *Desc* should be given as a list. The fields of an emission which are not specified are *Emission-ID, Next, Previous,* and *Time.* LispSEI will select an appropriate Emission-ID and insert the current world time.

The following functions currently in EFFECT.C either are complex (they involve some processing to implement, beyond simple data copying) or they are stubs representing future capabilities of the SEI. They will be removed and will at some later time be implemented in Lisp or properly implemented in C: **lightcycle, tempcycle, stop, move, turn, rkeat, emitodor, emitsound, eat, pickup, contact, near, drop, holding, nibble**

Control Commands

- **(Set-Debug** *flag*) sets the Debug flag to the specified value, either 0 or 1.

- **(Set-linkDebug** *flag*) sets the linkDebug flag to the specified value, either 0 or 1.

- (Start-SEI &optional (*debugfile* "seilog") (*started* 1)) calls the COMMUN.C function StartSei. *debugfile* is a string; *started* is an integer. *started* should be 0 if the SEI is started manually and then added to the world via a UI AttachSEI command; 1 otherwise. At the current time, an SEI is only Start-ed at the end of a master cycle, so nothing will appear to happen until all other ports in the world send Ok's to the master, at which time the entire world will be sent to the SEI.

- **(Enter-World** *worldName organismName* &optional (*debugfile* "seilog")) allows an SEI to join a world without requiring the oversight of a UI. All three arguments are strings. *worldName* should be the name of the world to join. *organismName* is the name to be associated with the SEI process (i.e., the mind as opposed to the body). This does not associate the organism with any object in the world. This may be done later via a Set-Organism-Body call. See the note for Start-SEI above regarding interaction with other ports in the world.

- **(Set-Organism-Body** *objNumber*) sets the organism body to be the object numbered *objNumber*, an integer.

- **(Save-State)** dumps the current Lisp process (organism + SEI) into the file specified in CKPfile in the directory specified by **(Path-Name)**.

- **(Ok-To-Proceed)** flushes the effector queue to the Master and sends an "OkToProceed" message.

- **(Receive-Updated-World)** waits for the master to send the updated world; it then absorbs the changes.

```
;;; --- This is Common Lisp --- ;;;

;;; World Modeler Sei Library-Definition package.
;;; Started 20-Oct-86 PHS.
;;; See /ml/usr/pshell/worldm/sei/seiprop.press or .mss for complete details.
;;; Uses lispsei.lisp, the base-level lisp SEI.

;;; This provides macros for defining sensors and filters.
;;; The top-level functions in this module are:

;;; (def-sensor <sensor-name> <sense-type> <object> <entity-filters>
;;;             <field-mungers>)

;;; (sense '<sensor-name>)

;;; (def-filter <name> <params> <sense-type> . <body>)

;;; (def-field-munger <name> <params> <sense-type> <field> . <body>)

;;; (def-constraint <name> <params> <sense-type> <function>)

;;; (Feel)
;;; (Listen)
;;; (Taste)
;;; (Smell)
;;; (See)

(defvar *TRACE-SENSE* NIL)

(defvar *SOUND-TYPE* 0)          ;;Equivalent to the defines in world.h
(defvar *SMELL-TYPE* 1)
(defvar *MAX-OBJECTS* 64)
(defvar *GOT-OBJECT-ARRAY* (make-array *MAX-OBJECTS* :element-type '(mod 2)))
;;; Bit N in the bit *GOT-OBJECT-ARRAY* is on iff object N has been retrieved
;;; from C in the current cycle.
(defvar *GOT-OBJECT-ARRAY-CYCLE* 0.0)
;;; Equal to the cycle that *GOT-OBJECT-ARRAY* was gotten on.
(defvar *OBJECT-ARRAY* (make-array *MAX-OBJECTS*))

(defvar *SENSES* '(vision hearing smell taste touch))
(defvar *ENTITIES* '(object sound smell taste contact))

;;; Wanked from LispSEI.l:
(defvar PRIMITIVE 0)
(defvar COMPLEXOBJ 1)
(defvar POLYGON 2)
(defvar CYLINDER 3)
(defvar CIRCLE 4)
(defvar SPHERE 5)
(defvar NOOBJECT 6)


;;; Like concat but expands what it can at expansion time and otherwise wraps
;;; a symbol-name call to function calls.
(defmacro smash (&rest seqs)
    \140(intern (concatenate 'string ,@(prepare-seqs seqs))))

(eval-when (load compile eval)
  (defun prepare-seqs (seqs)
    (mapcar #'prepare-seq seqs))

  (defun prepare-seq (seq)
    (cond ((numberp seq) (princ-to-string seq))
          ((listp seq) \140(princ-to-string ,seq))
          ((not (stringp seq)) \140(smart-symbol-name ,seq))
          (t seq)))

  (defun smart-symbol-name (thing)
    (cond ((or (numberp thing) (listp thing))
```

```
          (princ-to-string thing))
         ((not (stringp thing)) (symbol-name thing))
         (t thing))))

(defmacro putprop (obj val prop)
  \140(setf (get ,obj ,prop) ,val))



(defmacro Object-Object-Number (Object)
    \140(Object-Type-Object-Number ,Object))
(defmacro Object-Name (Object)
    \140(Object-Type-Object-Name ,Object))
(defmacro Object-Previous-Object (Object)
    \140(Object-Type-Previous-Object ,Object))
(defmacro Object-Next-Object (Object)
    \140(Object-Type-Next-Object ,Object))
(defmacro Object-Part-Of (Object)
    \140(Object-Type-Part-Of ,Object))
(defmacro Object-Part-Collision (Object)
    \140(Object-Type-Part-Collision ,Object))
(defmacro Object-Part-Magnetism (Object)
    \140(Object-Type-Part-Magnetism ,Object))
(defmacro Object-Transform (Object)
    \140(Object-Type-Transform ,Object))
(defmacro Object-Scale-Factor (Object)
    \140(Object-Type-Scale-Factor ,Object))
(defmacro Object-Location (Object)
    \140(Object-Type-Location ,Object))
(defmacro Object-Rotation (Object)
    \140(Object-Type-Rotation ,Object))
(defmacro Object-Velocity (Object)
    \140(Object-Type-Velocity ,Object))
(defmacro Object-Rotational-Velocity (Object)
    \140(Object-Type-Rotational-Velocity ,Object))
(defmacro Object-Acceleration (Object)
    \140(Object-Type-Acceleration ,Object))
(defmacro Object-Rotational-Acceleration (Object)
    \140(Object-Type-Rotational-Acceleration ,Object))
(defmacro Object-Forces (Object)
    \140(Object-Type-Forces ,Object))
(defmacro Object-Buoyancy (Object)
    \140(Object-Type-Buoyancy ,Object))
(defmacro Object-Mass-Center (Object)
    \140(Object-Type-Mass-Center ,Object))
(defmacro Object-Mass (Object)
    \140(Object-Type-Mass ,Object))
(defmacro Object-Enclosure (Object)
    \140(Object-Type-Enclosure ,Object))
(defmacro Object-Kind-Of-Object (Object)
    \140(Object-Type-Kind-Of-Object ,Object))
(defmacro Object-Composed-Of (Object)
    \140(Object-Type-Composed-Of ,Object))
(defmacro Object-Elasticity (Object)
    \140(Object-Type-Elasticity ,Object))
(defmacro Object-Restitution (Object)
    \140(Object-Type-Restitution ,Object))
(defmacro Object-Friction (Object)
    \140(Object-Type-Friction ,Object))
(defmacro Object-Inside-Color (Object)
    \140(Object-Type-Inside-Color ,Object))
(defmacro Object-Outside-Color (Object)
    \140(Object-Type-Outside-Color ,Object))
(defmacro Object-Outside-Surface-Texture (Object)
    \140(Object-Type-Outside-Surface-Texture ,Object))
(defmacro Object-Inside-Surface-Texture (Object)
    \140(Object-Type-Inside-Surface-Texture ,Object))
(defmacro Object-Taste (Object)
    \140(Object-Type-Taste ,Object))
```

```
;;; Only see things which are less than max-distance meters away from
;;; your eye.
(def-constraint object-distance-constraint (max-distance) vision
  (< (vector-distance (abs-object-location entity)
                      (abs-object-location sensorobj))
     max-distance))

;;; Only sense the smells that are near your "nose" (i.e., closer than
;;; max-distance meters away).
(def-constraint smell-distance-constraint (max-distance) smell
  (< (vector-distance (emission-origin entity)
                      (abs-object-location sensorobj))
     max-distance))


;;; Only sees objects which are bigger than object-enclosure meters.
(def-constraint object-size-constraint (object-enclosure) vision
  (> (object-enclosure entity)
     object-enclosure))

;;; Only smells smells which are stronger than smell-threshold.
(def-constraint smell-intensity-constraint (smell-threshold) smell
  (> (emission-intensity entity)
     smell-threshold))

;;; Only taste those tastes which are in contact with the sensor.
(def-constraint only-taste-touching () taste
  (member-if #'(lambda (contact)
                 (eq entity (object-taste (contact-object1-part contact))))
             (contacts-of sensorobj)))

;;; Only feel those contacts which are touching the sensor.  This is
;;; needed because contacts are just a list of all the contacts in the world.
(def-constraint only-my-contacts () touch
  (or (eq (contact-object1-part entity) sensorobj)
      (eq (contact-object2-part entity) sensorobj)))

(def-sensor sample-vision vision "organism"
  ((cone-of-vision (cos (/ pi 6))) disallow-heads) ())

;;;; -------------------  Sample Effector Commands --------------------;;;;

;;; Moves the organism forward (i.e., in the current direction) for one
;;; time unit with the given force (scalar, in Newtons).
(defun move-forward (force objnum)
  (let ((objdir (object-rotation (get-object objnum))))
    (effect-object-force
     (* force (vertex-x objdir))
     (* force (vertex-y objdir))
     (* force (vertex-z objdir))
     1.0
     objnum)))

(defun move-backward (force objnum)
  (let ((objdir (object-rotation (get-object objnum))))
    (effect-object-force
     (* force (- (vertex-x objdir)))
     (* force (- (vertex-y objdir)))
     (* force (vertex-z objdir))
     1.0
     objnum)))

;;; Applies the given torque (angular force) to object for one time unit.
;;; Note that torque = moment * angular acceleration, and moment is
;;; proportional to mass.
;;; First, find the current direction of the object.  Convert input
;;; magnitude and angle (in radians) to x,y.  Add to current direction
;;; and send torque command to master.
(defun turn-left (objnum magnitude &optional (angle (/ pi 20)))
```

```
  (let* ((object (get-object objnum))
         (objdir (object-rotation object))
         (dirx (vertex-x objdir))
         (diry (vertex-y objdir))
         (dirz (vertex-z objdir))
         (addx (* (cos angle) magnitude))
         (addy (* (sin angle) magnitude)))
    (effect-object-torque (+ dirx addx) (+ diry addy) dirz 1 objnum)))

(defun turn-right (objnum magnitude &optional (angle (/ pi 20)))
  (let* ((object (get-object objnum))
         (objdir (object-rotation object))
         (dirx (vertex-x objdir))
         (diry (vertex-y objdir))
         (dirz (vertex-z objdir))
         (addx (* (cos (- angle)) magnitude))
         (addy (* (sin (- angle)) magnitude)))
    (effect-object-torque (+ dirx addx) (+ diry addy) dirz 1 objrum)))
```

```
    (check-filters ',efilters entity)
    (check-mungers ',fmungers entity)
    (putprop ',sname ',stype 'sense-type)
    (putprop ',sname entity 'entity-type)
    (putprop ',sname filters 'filters)
    (putprop ',sname constraints 'constraints)
    (putprop ',sname ',fmungers 'fmungers)
    (putprop ',sname object 'sensorobj)
    (push ',sname (get ',stype 'sensors))))

(defmacro pp-sensor (sname &optional (verbose nil))
  (cond ((not (get sname 'entity-type))
         (format t "~S is not a sensor, sorry bud.~%" sname))
        (t
         (format t "~S sensor ~S: ~%" (get sname 'sense-type) sname)
         (format t "Attached to object: ~A~%" (get sname 'sensorobj))
         (maybe-format1 t "Filters: ~S~%" (get sname 'filters))
         (maybe-format1 t "Constraints: ~S~%" (get sname 'constraints))
         (maybe-format1 t "Field mungers: ~S%" (get sname 'fmungers))))
  t)

;;; Only format the single value if it is non-nil.
(defun maybe-format1 (stream control-string value)
  (if value
      (format stream control-string value)))

(defun print-entities (str entities etype)
  (format t str)
  (dolist (entity entities)
    (format t " ~A" (entity-name entity etype))))

(defun entity-name (entity etype)
  (ecase etype
    (OBJECT (object-name entity))
    (SOUND (smash "Sound " (emission-id entity)))
    (SMELL (smash "Smell " (emission-id entity)))
    (TASTE (smash "Taste " (list (taste-sweet entity) (taste-sour entity)
                                 (taste-bitter entity) (taste-salty entity))))
    (TOUCH (smash "Force " (apply-force entity)))))

;;; Returns the specifications of constraints in the fnames list.
(defun get-constraints (fnames)
  (let (res)
    (dolist (fname fnames)
      (if (constraintp (get-name fname))
          (push (eval-args fname) res)))
    (nreverse res)))

(defun get-filters (fnames)
  (let (res)
    (dolist (fname fnames)
      (if (filterp (get-name fname))
          (push (eval-args fname) res)))
    (nreverse res)))

;;; Checks to make sure that each of the names in fnames is either a filter
;;; or constraint, and for every one that isn't, tell the user.
;;; Also each one must be a filter of the appropriate entity type.
(defun check-filters (fnames entity)
  (dolist (fname fnames)
    (let ((real-fname (get-name fname)))
      (cond ((and (not (filterp real-fname))
                  (not (constraintp real-fname)))
             (format t
                     "%~S is not a constraint or filter, Def-sensor will ignore it.~%"
                     real-fname))
            (t (check-type&number fname real-fname entity))))))

(defun check-mungers (fnames entity)
```

```
  (dolist (fname fnames)
    (let ((real-fname (get-name fname)))
      (cond ((not (mungerp real-fname))
             (format t
                     "%~S is not a munger, Def-sensor will ignore it.~%"
                     real-fname))
            (t (check-type&number fname real-fname entity))))))

;;; PRECONDITION: NAME names a defined filter, constraint or field-munger.
(defun check-type&number (spec name entity)
  (cond
   ((not (eq (get name 'entity-type) entity))
    (error "~S must be a filter for entity type ~S but is not.~%"
           entity))
   ((not (= (get-num-params spec)
            (get name 'num-params)))
    (error "~S must be given ~S params but was only given ~S.~%"
           name
           (get name 'num-params)
           (get-num-params spec)))))

;;; Since filter names, constraint names and object names may be lists with
;;; additional parameters.
(defun get-name (fname)
  (if (atom fname)
      fname
      (car fname)))

(defun get-num-params (fname)
  (if (atom fname)
      0
      (length (cdr fname))))

;;; If fname has arguments, then replace each argument by that argument
;;; evaluated.  For example, if the user gives a (cos (/ pi 4)) arg to a
;;; filter fn by writing (my-filter (cos theta)), this fn would change that
;;; to (my-filter 0.7071067) and return it.
(defun eval-args (fname)
  (if (atom fname)
      fname
      (do ((args (cdr fname) (cdr args)))
          ((null args) fname)
        (rplaca args (eval (car args))))))

(defun constraintp (name)
  (get name 'constraint))

(defun filterp (name)
  (get name 'filter))

(defun mungerp (name)
  (get name 'munger))

(defmacro def-filter (name params sense-type &rest body)
  (when (or (get name 'constraint)
            (get name 'munger))
    (cerror "Re-define it as a filter"
            "~S is already defined as a constraint or munger." name)
    (putprop name nil 'constraint)
    (putprop name nil 'munger))
  (let ((entity (sense-to-entity sense-type)))
    \140(progn
      (format t "Defining filter ~S~%" ',name)
      (putprop ',name ',entity 'entity-type)
      (putprop ',name t 'filter)
      (putprop ',name ,(length params) 'num-params)
      (defun ,name (entities sensorobj ,@params)
        ,@body))))
```

```
(defmacro def-constraint (name params sense-type &rest body)
  (when (or (get name 'filter)
            (get name 'munger))
    (cerror "Re-define it as a constraint"
            "~S is already defined as a filter or munger." name)
    (putprop name nil 'filter)
    (putprop name nil 'munger))
  (let ((entity (sense-to-entity sense-type)))
    \140(progn
      (format t "Defining constraint ~S~%" ',name)
      (putprop ',name ',entity 'entity-type)
      (putprop ',name t 'constraint)
      (putprop ',name ,(length params) 'num-params)
      (defun ,name (entity sensorobj ,@params)
        ,@body))))

;;; When the given field is accessed for the given type of entity,
;;; pre-process the value with the code in <body>.
(defmacro def-field-munger (name params sense-type field &rest body)
  (when (or (get name 'filter)
            (get name 'constraint))
    (cerror "Re-define it as a munger"
            "~S is already defined as a filter or constraint." name)
    (putprop name nil 'filter)
    (putprop name nil 'constraint))
  (let ((entity (sense-to-entity sense-type)))
    \140(progn
      (format t "Defining munger ~S~%" ',name)
      (putprop ',name ',field 'field)
      (putprop ',name ',entity 'entity-type)
      (putprop ',name t 'munger)
      (putprop ',name ,(length params) 'num-params)
      (defun ,name (field) ,@params)
        ,@body)))
```

```
;;;; The biggee.

;;; For now, fields are munged only after the filters are executed.  This
;;; makes things much easier.
;;; Algorithm:
;;; (1) get initial list.
;;; (2) take out those entities which don't meet all the constraints.
;;; (3) pass the entities list through all the filters.
;;; (4) munge the appropriate fields
;;; (5) return the result.
(defun sense (sname)
  (let ((etype (get sname 'entity-type)))
    (if (null etype)
        (error "Sorry, ~S is not a sensor defined by def-sensor.~%" sname))
    (ecase etype
      (OBJECT (format t "Looking...~%"))
      (SOUND (format t "Listening...~%"))
      (SMELL (format t "Sniffing...~%"))
      (TASTE (format t "Tasting...~%))
      (CONTACT (format t "Feeling...~%")))
    (let ((entities (make-initial-entity-list etype))
          (sensorobj (get-object-named (get sname 'sensorobj))))
      (if *TRACE-SENSE* (print-entities "Initial list is: " entities etype))
      (when (null entities)
        (cerror "Send it through the filters anyway.~%"
                "Sensor ~S sensed nothing.~%" sname))
      (setq entities (constrain-entities entities sname sensorobj))
      (setq entities (filter-entities entities sname sensorobj))
      (munge-fields entities sname)
      entities)))

;;; For every filter in the filter list associated with the given sensorobj,
;;; pass the entities list through it.
(defun filter-entities (entities sname sensorobj)
  (let ((fname nil)
        (fparams nil)
        (etype (get sname 'entity-type)))
    (dolist (filter (get sname 'filters))
      (cond ((atom filter)
             (setq fname filter fparams nil))
            (t (setq fname (car filter) fparams (cdr filter))))
      (when *TRACE-SENSE*
        (format t "Calling filter ~S" fname)
        (if fparams
            (format t " with arguments ~S.~%" fparams)
            (format t ".~%")))
      (setq entities (apply fname \140(,entities ,sensorobj ,@fparams)))
      (if *TRACE-SENSE* (print-entities "List is now: " entities etype))))
  entities)

;;; For each entity in the entities list, it must meet all the constraints
;;; associated with the sensor sname to stay in the result.
(defun constrain-entities (entities sname sensorobj)
  (let ((cname nil)
        (cparams nil)
        (etype (get sname 'entity-type)))
    (dolist (constraint (get sname 'constraints))
      (cond ((atom constraint)
             (setq cname constraint cparams nil))
            (t (setq cname (car constraint) cparams (cdr constraint))))
      (when *TRACE-SENSE*
        (format t "Calling constraint ~S" cname)
        (if cparams
            (format t " with arguments ~S.~%" cparams)
            (format t ".~%")))
      (let ((rem-params (cons sensorobj cparams)))
        (do ((entity entities (cdr entity))
             (laste nil entities))
            ((null entity) entities)
```

```
          (cond ((not (apply cname (cons (car entity) rem-params)))  ;;constrained
                 (if *TRACE-SENSE*
                     (format t "Entity ~S gets constrained.~%"
                             (entity-name (car entity) etype)))
                 (if (null laste)
                     (setq entities (cdr entities))
                     (setf (cdr laste) (cdr entity))))))
    (if *TRACE-SENSE* (print-entities "List is now: " entities etype))
    entities))

;;; Won't work for all field types since LispSEI doesn't provide setting
;;; functions for all field types.
(defun munge-fields (entities sname)
  (let ((mname nil)
        (mparams nil)
        (etype (get sname 'entity-type)))
    (dolist (munger (get sname 'fmungers))
      (cond ((atom munger)
             (setq mname munger mparams nil))
            (t (setq mname (car munger) mparams (cadr munger))))
      (let ((field (get mname 'field)))
        (when *TRACE-SENSE*
          (format t "Munging field ~S with ~S" field mname)
          (if mparams
              (format t " with arguments ~S.~%" mparams)
              (format t ".~%")))
        (dolist (entity entities)
          (set-field etype field entity
                     (funcall mname (cons (get-field mname field) mparams)))))))))

(defun make-initial-entity-list (etype)
  (ecase etype
    (OBJECT (make-object-list))
    (SOUND (make-sound-list))
    (SMELL (make-smell-list))
    (TASTE (make-taste-list))
    (CONTACT (make-feel-list))))

;;; Make and return a list containing all the objects in the world.
(defun make-object-list ()
  (make-object-list0 (object-head)))

(defun make-object-list0 (objnum)
  (if (not (zerop objnum))
      (do* ((topobj objnum (object-next-object object))
            (object (get-object topobj) (get-object topobj))
            (res (cons object (make-object-list0
                                (object-composed-of object)))
                 (cons object (nconc (make-object-list0
                                       (object-composed-of object)) res))))
           ((zerop (object-next-object object)) res))))

(defun make-feel-list ()
  (contacts))

;;; Make and return a list containing all the emissions of type sound in the world.
(defun make-sound-list ()
  (do ((emitnum (emission-head) (get-emission-next emission))
       (emission nil (emission emitnum))
       (res nil (if (soundp emission)
                    (cons emission res)
                    res)))
      ((null emitnum) res)))

(defun make-smell-list ()
  (do ((emitnum (emission-head) (get-emission-next emission))
       (emission nil (emission emitnum))
       (res nil (if (smellp emission)
                    (cons emission res)
```

```
                    res)))
      ((null emitnum) res)))

(defun soundp (emission)
  (= (get-emission-emission-type emission) *SOUND-TYPE*))

(defun smellp (emission)
  (= (get-emission-emission-type emission) *SMELL-TYPE*))

;;; Just like make-object-list except returns the taste fields instead of
;;; the whole objects.
(defun make-taste-list ()
  (make-object-list0 (object-head)))

(defun make-taste-list0 (objnum)
  (if (not (zerop objnum))
      (do* ((topobj objnum (object-next-object object))
            (object (get-object topobj) (get-object topobj))
            (res (list object)
                 (cons (object-taste object)
                       (nconc (make-object-list0
                                (object-composed-of object)) res))))
           ((zerop topobj) res))))

;;; Return a list containing all the lists of entities sinsed by each
;;; sensor of the FEEL type.
(defun feel ()
  (get-all-sensory-input 'touch))

(defun hear ()
  (get-all-sensory-input 'hearing))

(defun taste ()
  (get-all-sensory-input 'taste))

(defun smell ()
  (get-all-sensory-input 'smell))

(defun see ()
  (get-all-sensory-input 'vision))

(defun get-all-sensory-input (sense)
  (mapcar #'(lambda (sensor)
              (sense sensor))
          (get sense 'sensors)))
```